

# Data Structure and Integration in Program

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Various data structures.
- Difference between primitive and non-primitive data structures.
- Difference between tree and graph.
- Various tree data structures.
- Binary tree and how it is useful.
- Best data structure for your needs.

## 17.1 | Introduction

Data structures are essentially arrangements that can be used for storage and manipulation of the internal data of a computer program. What is interesting to note here is that most novice developers and programmers of Java programming language start using data structures long before they even know what they really are or how they are different from other data types.

While there are a lot of different data structures, some of the most commonly used ones include stacks linked lists and arrays. Having a strong understanding of the different types of data structures available in Java language and how they differ from each other is imperative for a number of reasons. The choice of data structure does not only affect the time taken by the application to perform crucial tasks; however, the choice of data structure has a strong connection with the effort that is required for implementation and the performance of the block of code, program, or application.

Since you now understand why it is important to know the difference between each of the data structures that are available, we will discuss a number of different data structures in great detail, take a look at their advantages and disadvantages, and explain how each of them can be used in programs.

## 17.2 | Introduction to Data Structures



Data structures are closely related to **abstract data types (ADTs)**, which is another extremely important area in Java programming language. Best termed as a unique mathematical model for data types, an ADT is essentially defined by the user based on how it is expected to act in the program, block of code, or application where it is to be used, and the operations that are likely to be performed on it.

While data structures are essentially based on ADTs, the difference lies in the fact that there is a concrete implementation mechanism in place for data structures, and they cannot be arbitrarily used like ADTs.

As complex and beneficial as they are for efficient programming that gives optimal results, data structures can be classified and categorized into a number of different groups, each of which has its own characteristics, features, and properties. The focus of this chapter will be exclusively on data structures, their types, and their correct usage.

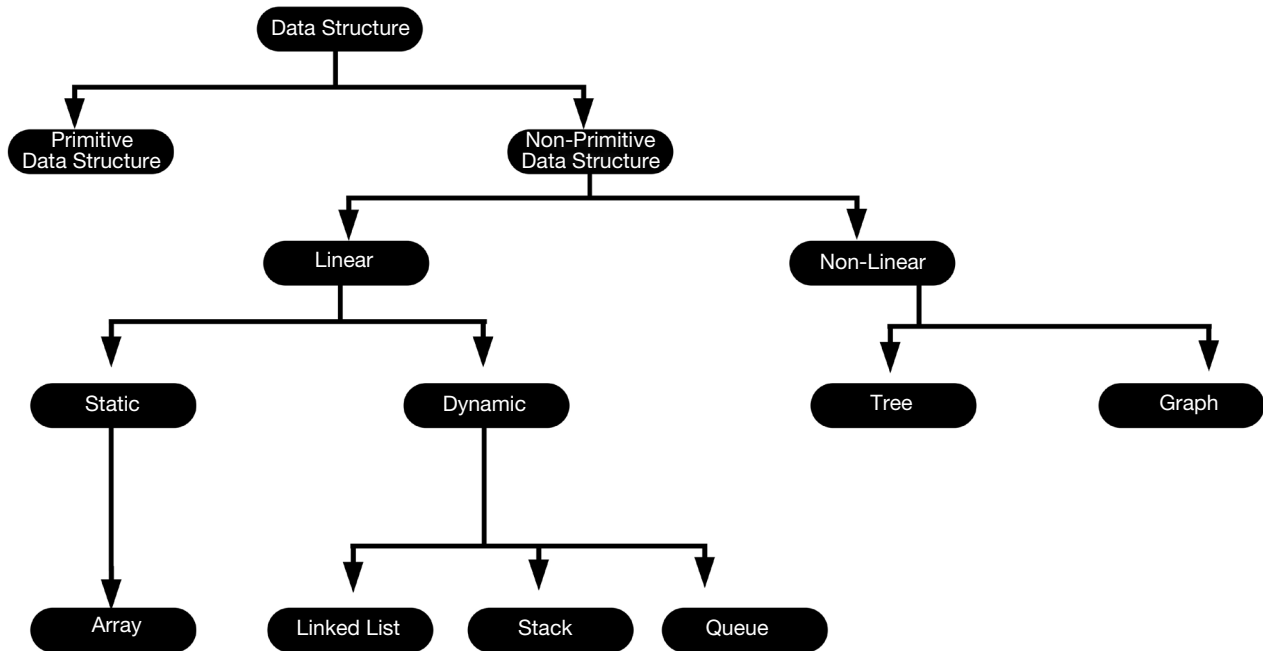


How do you program without any data structure?



## 17.3 | Classification of Data Structures

As mentioned, there are a number of different ways in which data structures of Java language can be classified. The most common classifications are primitive data structures and non-primitive data structures. Figure 17.1 shows the data structure hierarchy.



**Figure 17.1** Data structure hierarchy.

### 17.3.1 Primitive Data Structures

Primitive data structures are nothing but the basic data types that are available in every programming language. They are used for the most fundamental of reasons. These predefined ways of storing data in Java language come with a limited set of operations that can be performed on them, essentially setting a limit to the number and types of situations for which they can prove to be beneficial.

We have learned about this in the Chapter 11, but let us revise it again. There are eight different primitive data structures available in Java language, namely, int, char, float, Boolean, long, double, short, and byte. Often referred to as the fundamental building blocks for the manipulation of data in Java, these data structures can be used to store the simplest types of values – that too of a single kind. Since primitive data types are very limited in what they offer, it is no surprise that there are certain situations where primitive data structures alone do not suffice. And that is where derived data structures come into the picture.

Derived data structures and user-defined data structures not only offer a much wider range of applications, but their benefits are also unique in a number of ways. We will learn about non-primitive data structures and how they differ from their primitive counterparts in the following sections. For now, we will define each of the primitive data structure of Java language along with examples of how primitive data structures can be declared and used in any program, application, or block of code, and what they can be used to achieve.

1. **Byte:** The Byte primitive is one of the basic predefined data structures of Java language, which is found in the form of an 8-bit signed two's complement integer. The Byte primitive has a maximum value of 127, and can take values as low as  $-128$ . The default value of this data structure is set as 0.

The fact that the Byte primitive requires far less storage space than the int type is one of the major reasons why it is so popular among developers. A Byte is four times smaller than an integer, which is why it can be easily used in certain scenarios in place of integers, especially when storage space is a concern.

Here is an example of how a Byte primitive can be declared:

```
byte a = -37;
```

2. **Short:** Like Byte, the Short primitive is also a two's complement signed integer. The difference between the two, however, lies in the fact that the Short data structure has 16-bit values. What this means is that the range of values for Short is much wider than for Byte, making the maximum possible value for Short 32,767 and the minimum possible value -32,768.

Like the Byte primitive, the Short primitive also has a default value of 0, and is used especially in situations where storage is a concern. This is because the Short type takes two times less space than the integer type.

Here is an example of how the Short primitive can be declared to be used in any program written using Java language:

```
short s1 = 32000;
```

3. **Int:** A signed two's complement integer, the Int primitive allows a 32-bit value, making the range of the data structure much larger than both the Short and Byte primitives combined. Since the primitive allows 32-bit values, the maximum value that can be stored in an Int type is  $2^{31} - 1$ , or 2,147,483,647, whereas the minimum value that is allowed in the Int type is -2,147,483,648. The default value for the Int type is 0.

Even though the Int type takes up a lot more memory than the Short and Byte types, it is often the data structure of choice for integer values, unless memory or storage space is a concern.

Here is an example of how an Int type variable can be declared in Java language:

```
int a = 500000;
int b = -500000;
```

4. **Long:** The Long primitive of Java language is a signed two's complement integer that can have a 64-bit value. Needless to say, the range of values allowed by Long is far wider than all other primitives in Java, with possible values ranging from a minimum of  $-2^{63}$ , or -9,223,372,036,854,775,808, to a maximum of  $2^{63} - 1$ , or 9,223,372,036,854,775,807.

Since this primitive takes up more memory and requires more storage space than variables of the Int type, the Long primitive is only used for integer values that will not be possible with the Int primitive. Additionally, values of the Long primitive are easily differentiable from values of variables of other primitives because they are always terminated with L. Similar to the default values of other variables discussed above, the default value for the Long primitive is 0L.

Here is an example of how a variable of the Long type can be declared in Java language:

```
long a = 19823290832L;
```

5. **Float:** The Float data structure in Java language is an extremely interesting and useful one for many reasons. The Float primitive allows single precision 32-bit values, and is primarily used for saving memory in situations where arrays of floating point numbers need to be used. Like the values of the Long type, values of the Float primitive are also easily differentiable thanks to the "f" at the end of all values of the data type.

The default value of the Float primitive is 0.0f, and the data type is never used for situations in which precise values are needed.

Here is an example of how a variable of the float type can be declared in Java language:

```
float f1 = 2.5f;
```

- 6. Double:** As the name suggests, the Double primitive is a double precision data structure in Java language that supports 64-bit values, giving it a wider range of possible values than the float data structure. This primitive is generally used as the default choice of programmers when they need to deal with decimal values.

Like the Float data structure, the Double primitive should never be used in situations where accuracy and precise values are imperative for the integrity of the program or application to be maintained. Additionally, values of Double variables are also easily differentiable as they end with a “d”.

Here is an example of how a variable of the Double primitive can be declared in Java language:

```
double d1 = 234.5;
```

- 7. Boolean:** The Boolean primitive is used for representation of a single bit of data in Java language. Unlike the other primitives and the range of values that were allowed in them as discussed above, the Boolean primitive allows only two values – false and true.

The only area where this primitive can be used is for tracking whether a condition will be true or false. The default value for variables of the Boolean type is set as false.

Here is an example of how a variable of the Boolean primitive can be declared in Java language:

```
boolean var = true;
```

- 8. Char:** The Char primitive is used for representation of a single 16-bit Unicode character in Java language. The range of values allowed by the Char primitive is rather limited, with the minimum possible value as 0, or “\u0000”, and the maximum possible value 65,535, or “\uffff”.

The Char primitive is only used in situations where a single character needs to be stored.

Here is an example of how a variable of the Char primitive can be declared in Java language:

```
char c1 = 'b';
```

As evident from the explanation of each of the primitives and the range of values permissible by each of them, it goes without saying that another mechanism for the storage of variables and their manipulation was crucial. Primitives of Java language are not only limited in the operations that can be performed on them, but the range of values allowed by them also limits the possibilities.

And this is exactly why non-primitive data structures such as arrays, stacks, and linked lists exist.

While they are essentially made up of primitive data structures, non-primitive data structures provide a lot more room for operations, and leave users with more to play around.

In the following section, we will learn about the most common non-primitive data types and explain how each of them can be used, their advantages, and how they differ from each other.

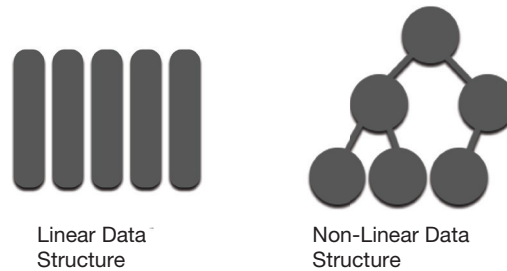


What is the difference between Float and Double?

### 17.3.2 Non-Primitive Data Structures

While non-primitive data structures are based on their primitive counterparts, the range of functionality offered by non-primitive data structures and the benefits that they offer are greater than primitive data structures. As shown in Figure 17.2, non-primitive data structures can be classified into two major groups – linear non-primitive data structures

and non-linear non-primitive data structures. Table 17.1 lists the differences between linear and non-linear non-primitive data structures.



**Figure 17.2** Representation of linear and non-linear data structures.

**Table 17.1** Linear data structure vs non-linear data structure

Parameter	Linear Data Structure	Non-Linear Data Structure
<b>Basic</b>	Elements are arranged adjacent to each other	Elements are arranged in a sorted order
<b>Traversing of the data</b>	Data elements are traversed in one go	Data elements cannot be traversed in one go
<b>Ease of implementation</b>	Simple	Complex
<b>Levels involved</b>	Single	Multiple
<b>Examples</b>	Array, LinkedList, Queue, Stack, etc.	Graph, Tree
<b>Memory utilization</b>	Inefficient	Efficient

**QUICK CHALLENGE**

Define a situation in which you could use non-linear data structure.

### 17.3.2.1 Linear Non-Primitive Data Structures

For a data structure to be considered as linear, it is imperative for the elements that make it should form a linear list. Another requirement is that the constituent elements of the data structure should be connected to each other adjacently and follow a certain order. Additionally, for a data structure to be considered linear, the memory that it consumes should also be in linear fashion, and the storage of elements in the memory must be sequential.

Linear non-primitive are also differentiable from their non-linear counterparts because a certain amount of memory has to be declared in advance for the linear data structure, unlike in the case of non-linear data structures. While this requirement often results in loss of memory and mismanagement due to improper utilization of memory, doing so is imperative if you want to use linear data structures. Additionally, since memory and element storage for linear data structures works in a linear and sequential fashion, constituent elements of a linear structure can only be reached sequentially, and only a single element of the data structure can be visited without a reference. Since it is imperative for adjacency relationships to be maintained for data structures, the operations that can be performed on linear data structures are also limited in a number of ways. What this means is that it is not possible for one to insert or delete constituent elements of a linear data structure arbitrarily, and all operations must be performed in a sequential fashion. Here are some of the operations that can be performed on linear data structures in Java language:

1. Addition of elements.
2. Deletion of elements.
3. Traversing the data structure.

4. Sorting the constituent elements of the data structure.
5. Searching for a constituent of the linear data structure.

Now, we will look at a few of the most popular linear data structures in detail, and learn how they can be used in programs or applications.

### 17.3.2.1.1 Arrays

Arrays are perhaps the most popular data structure in Java and other programming languages. Often built into programming languages, arrays act as a great starting point to not only introduce novice programmers to the concept of data structures, but also to understand how object-oriented programming (OOP) and data structures go hand in hand. In this section, we will learn how to create arrays from scratch, use them effectively and efficiently, and perform operations on them. This will help users get an idea of how data structures really work; we will then move on to more complicated linear and non-linear data structures that are used in Java language.

Arrays in Java are created dynamically and can contain multiple elements that essentially define the size or length of the array. What this means is that if an array contains  $x$  elements, the length of the array will be  $x$ .

While all of the constituent elements of an array have the same name, they each have a unique reference based on where they lie in the linear sequence of the data structure. To reference any particular element in the data structure, you will need the index of the element in the array, which will always be in the form of a non-negative integer. Since the index of the first element is always 0, the last constituent element of any particular array will always be  $x - 1$ , where  $x$  is the length, size, or total number of elements present in the data structure.

It is important to note that even though it was mentioned that an array can contain multiple elements, the condition is that all of these elements must be of the same type, which is often called the component type of an array. What this means is that even if an integer array has been created, you will not be allowed to enter float values in the array. If an integer array needs to be created as in the case mentioned above, we will declare the integer array as follows:

```
int age[]; // This line of code declares an integer array to hold ages
```

It is also important to note that arrays in Java can have multiple dimensions, but that does not change the fact that there are no restrictions on the element type or component type of the array. Additionally, whenever a variable of the array type is created, it does not set aside any amount of memory or create the object for the array. Instead, the only thing that really happens is the creation of the variable for the array – a variable that may be used to contain the reference to the array itself.

Now that you have a basic understanding of how arrays work, their characteristics, and the operations that can be performed on them, we will focus on the practical implementation of arrays and how that can be done in Java.

**Declaration of simple arrays:** An array in Java language is essentially a list of elements that have the same type and are referred to by the same name. As seen above, there is not much that needs to be done to declare an array in Java besides adding a pair of square brackets after the name of the array that you wish to create.

Another way to do this is by adding the pair of square brackets after the type of array, as follows:

```
float[ ] temperature;
```

Therefore, the permissible syntax for the declaration of an array is as follows:

```
type[ ] name;
```

or

```
type name[ ];
```

As mentioned above, simply declaring an array does not suffice. Instead, you will have to create or initialize your array for you to be able to effectively use it in the program, application, or block of code. Next, we will explain how an array can be created, and how memory can be allocated to ensure that the array that was declared is functional and can be used in the program.

**Creating a one-dimensional array:** Whenever an array is to be created to be used in Java, the **new** keyword will be used, and the length or size of the array will be specified in square brackets. Here is an example of how this will work:

```
Int age[ ]; // this line of code declares a one-dimensional array of the integer type named age
```

```
age = new int[4]; // this line of code creates a new array after declaring it
```

**Initialization of a one-dimensional array:** This is perhaps the easiest and simplest part of the process. The only thing that needs to be done for initialization of an array in Java is to place values in curly braces separated by commas. Here is how this can be done:

```
int age[4] = {11,22,35,64};
```

Now that you understand how an array can be declared, created, and initialized in Java, here is a sample code that will help put things into perspective:

```
package javall.fundamentals.chapter17;
public class ArrayExample {
    public static void main(String args[]) {
        // Following code declares an int array
        int myIntArr[] = { 1, 5, 993, 35 };
        // Following code iterates through array elements and print them one by one
        for (int b = 0; b < myIntArr.length; b++) {
            System.out.println(myIntArr[b]);
        }
    }
}
```

The above program produces the following result.

```
1
5
993
35
```

As seen above, the declaration, creation, and initialization of arrays in Java language is extremely easy and simple. However, there are advantages and disadvantages to the concept of arrays in Java language.

Although arrays make for a great data structure and provide a resourceful mechanism for the storage and manipulation of large amounts of data, the major problem is that they can only be used to store data elements of a single type. Moreover, once an array has been declared, it is not possible for it to be changed in any way. What this means is that once you have declared an array, it will not be possible to increase or decrease its size, resulting in memory wastage.

**QUICK CHALLENGE**

Define two arrays of same size and integer type. Write a program to create a third array which contains the values which are the result of multiplication of each element from the first array and second array. [The first element of first array will get multiplied with the first element of second array, the second element of first array will get multiplied with the second element of second array, and so forth.]

### 17.3.2.1.2 Lists and Queue

The List interface in Java language is particularly interesting for a number of reasons. While there are several implementations of the List interface, they all work on an identical principle. Much like arrays, elements in Lists of Java can also be accessed according to their index or position in the list. The same mechanism is also used for the addition of elements, and duplication is possible within lists as well. Additionally, since List extends Collection in Java language, all of the operations of Collection are supported by List as well.

Queue is another data structure which works on the First In, First Out (FIFO) principal. We have covered these topics in detail in Chapter 13.

### 17.3.2.2 Non-Linear Non-Primitive Data Structures

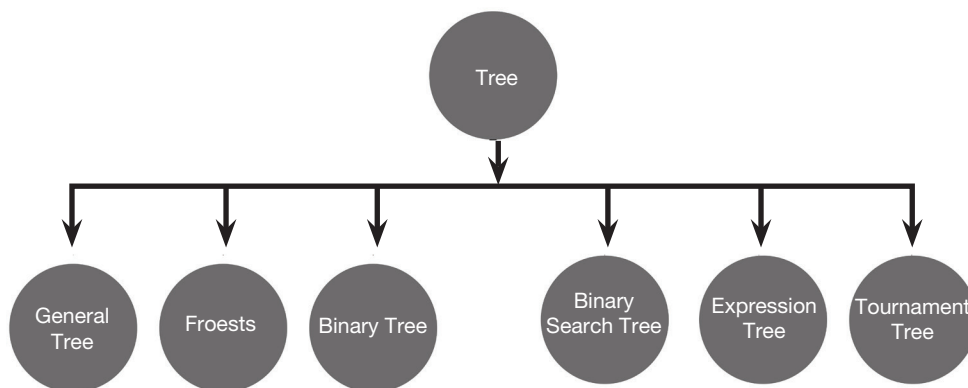
The following data structures come under the non-linear non-primitive data structures category. Data in this structure is not arranged sequentially but in a sorted order. In this type of structure, all the operations are done in a non-sequential manner such as traversal of data elements, insertion, and deletion.

These types of non-linear data structures are memory efficient, as they use memory resourcefully and do not need pre-memory allocation. The two types of data structures available are *tree* and *graph*. Data is arranged in a hierarchical relationship in the tree structure, which involves the relationship between the child, parent, and grandparent.

#### 17.3.2.2.1 Tree Data Structure

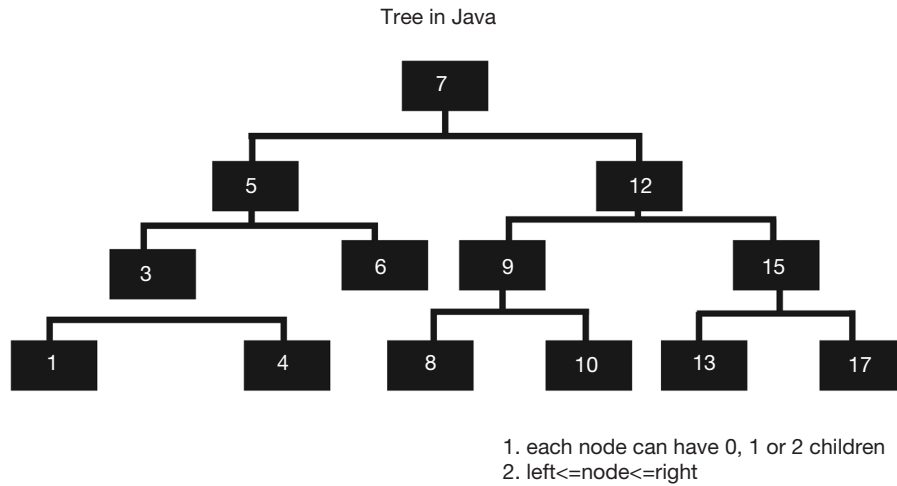
As shown in Figure 17.3, the following are tree related data structures:

1. General tree.
2. Forests.
3. Binary tree.
4. Binary search tree.
5. Expression tree.
6. Tournament tree.



**Figure 17.3** Tree data structures.

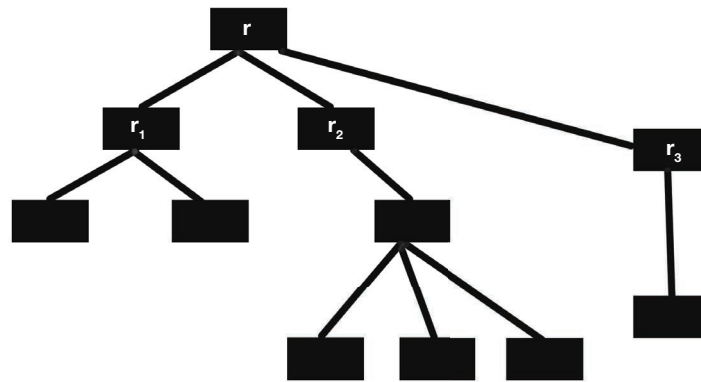
Most of the time, you will be using binary tree and binary search tree data structures. These are two of the popular data structures commonly used to solve many problems. Figure 17.4 shows the tree representation in Java.



**Figure 17.4** Tree representation.

- General trees:** Many times we encounter a set of data which is not in a format that can be handled by binary tree. Binary tree can have maximum 2 nodes. Hence, cases like displaying organization hierarchy cannot be handled by binary tree as the CEO (root node) can have more than 2 vice presidents under it. For this type of data, we need a tree data structure which can handle multiple nodes under any root node. This data structure is known as general tree, which is shown in Figure 17.5.

A general tree  $T$  is a tree which has one root node  $r$  and finite set of one or more nodes. Tree  $T$  starts with root node  $r$ . If the first set  $(T - \{r\})$  is not empty, then the rest of the nodes are divided into  $n \geq 0$  disjoint subset trees like  $T_1, T_2, \dots, T_n$  which are called *subtrees*. Each of these trees has a root node which looks like  $r_1, r_2, \dots, r_n$ , respectively. These root nodes are children of  $r$ .

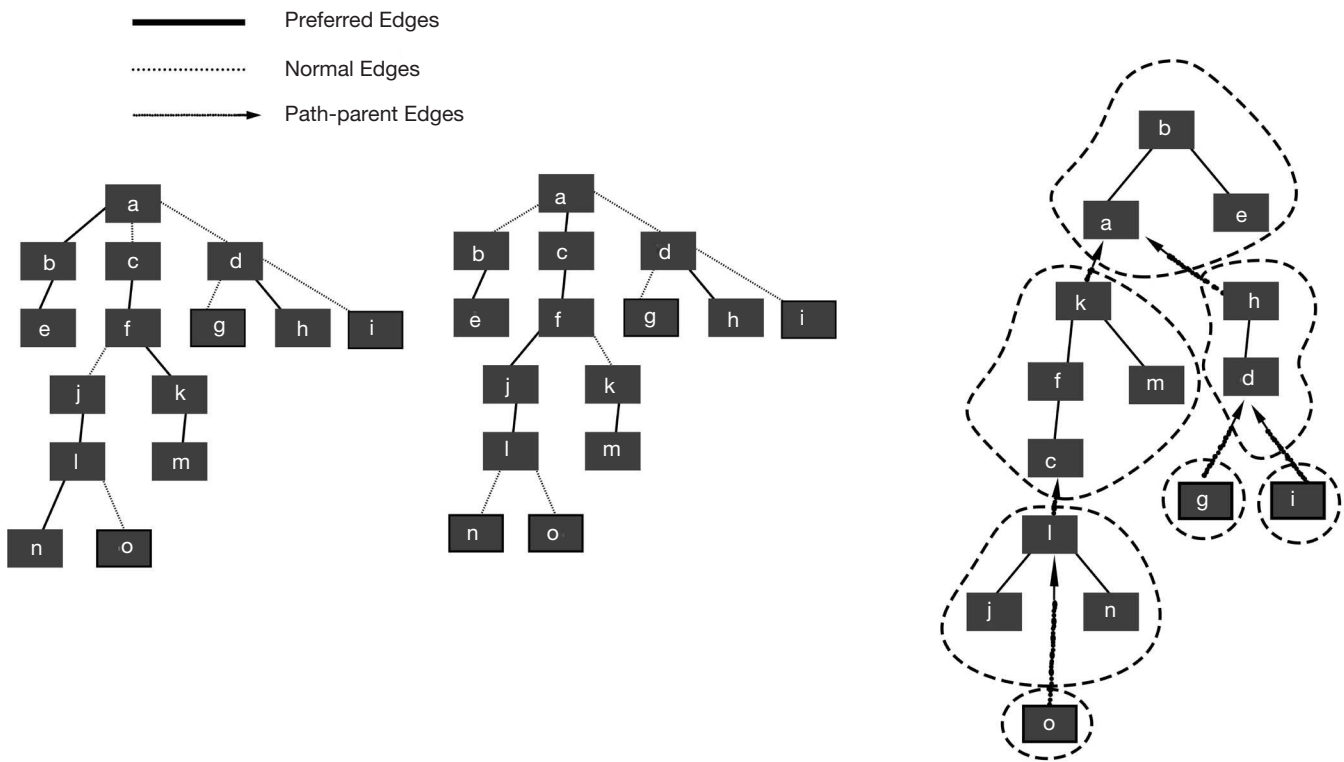


**Figure 17.5** General tree structure.

**QUICK  
CHALLENGE**

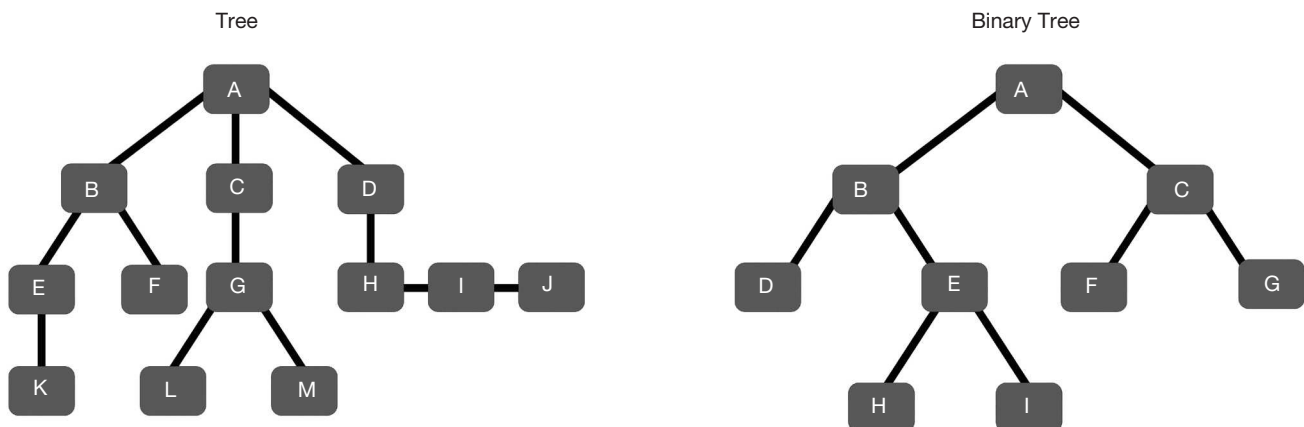
Give an example of a real-life scenario which is suitable to be represented as a general tree.

- Forests:** An arbitrary set of trees is called Forest. Figure 17.6 shows representation of Forests. A Forest can start with one root node; it then grows in an ordered fashion, with one node having any number of child nodes. The children of a node are sequenced as first, second, and so on. Contrary to the binary tree, this tree does not have a concept of left and right. However, we can sequentially draw this tree from left to right. An Ordered Forest is nothing but an ordered set of ordered trees, which are also termed as Orchard.



**Figure 17.6** Representation of Forests.

3. **Binary tree:** As we already know, tree data structure is hierarchical in nature. Binary Tree is a subset of Tree in which each node has at most two children. Figure 17.7 shows Tree vs Binary Tree. These children are referred as the left and right child. This tree is implemented using links. The topmost node is the pointer in the tree. If tree is empty, then the value of root is NULL.



**Figure 17.7** Representation of Tree vs Binary Tree.

A node has three parts: (a) data, (b) pointer to left child, and (c) pointer to right child.

There are two ways in which a binary tree can be traversed:

- (a) **Depth-first traversal:** In this case, there are three ways to traverse:
  - In the first way, it starts with left node, then root node, and then right node is accessed. This is called *inorder (Left-Root-Right) traversal*.
  - In the second way, it starts with the root node, then moves to the left node, and then comes to the right node. This type of traversal is called *preorder (Root-Left-Right) traversal*.
  - The third way is to start with left node, then move to the right node, and finally come to the root node. This is called *postorder (Left-Right-Root) traversal*.
- (b) **Breadth-first traversal:** In this case, the traversal takes place level-by-level. As name suggests, the focus is on the breadth of the level. In other words, at every level, the traversal takes place on the full width of the tree before going to the next level. Let us take an example of the binary tree shown in Figure 17.7. In this example, node A is at level 1, nodes B and C at level 2, nodes D, E, F, and G are at level 3, and nodes H and I are at level 4. In this case, the algorithm will first traverse through node A, then through nodes B and C, then nodes D, E, F, G, and finally it will traverse through nodes H and I.

Following are some of the properties of binary tree:

- (a) The maximum number of nodes at level  $x$  of a binary tree is always  $2^{x-1}$ .
- (b) The maximum number of nodes of height  $y$  is  $2^{y-1}$ .
- (c) The maximum number of levels or height in tree with  $n$  nodes is  $\log_2(n+1)$ .
- (d) Every node in the tree has 0 or 2 children.
- (e) Total number of nodes with 2 children are always less by one than the number of leaf nodes (i.e.,  $L = X + 1$ , where  $L$  is number of leaf nodes and  $X$  is internal nodes with two children).



Construct a binary tree for the following array:

`myArray[] = {1, 2, 3, 4, 5, 6}`

4. **Binary search tree:** This tree is similar to the Binary Tree. It is mainly used for faster search over LinkedList, but it is slower than arrays. For insertion and deletion, it is better than arrays but not LinkedList. This is an important data structure that you should be aware of. It provides efficient search with  $O(\log n)$  complexity.



Can a Binary Search Tree's topmost node contain a null node?

It has three more other properties than Binary Tree.

- (a) It contains keys less than the node's key on the left side.
- (b) It contains keys greater than the node's key on the right side.
- (c) Both sides, left and right, must be a Binary Search Tree.

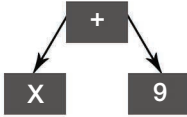
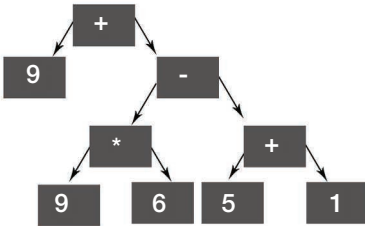
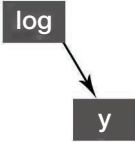
A Binary Search Tree is quite useful in applications such as e-commerce, where products are getting added or deleted from the inventory constantly and then presented in a sorted order.



Can a Binary Search Tree's topmost node contain more than two elements?

5. **Expression tree:** As name suggests, an Expression Tree is one made up of expression, wherein each internal node contains an operator and each leaf node contains an operand. For example, in the expression  $x + 9$ ,  $x$  and 9 are operands and  $+$  is the operator. Hence, the internal node will start with  $+$ , and on the left we will have  $x$  and on right we will have 9. Table 17.2 shows a few examples of Expression Trees.

**Table 17.2** Examples of Expression Trees

Expression	Expression Tree	Inorder Traversal Result
$(x+9)$		$x+9$
$9+(9*6-(5+1))$		$9+9*6-5+1$
$\log(y)$		$\log(y)$

6. **Tournament trees:** A complete Binary Tree with  $n$  external nodes and  $n - 1$  internal nodes is called Tournament Tree. All the external nodes are characterized as players and all the internal nodes are characterized as winners. Winner node is situated between the two external nodes termed as players. As we have just discussed, a Tournament Tree gets  $n - 1$  internal nodes and  $n$  external nodes. Thus, to find the winner, we need to eliminate  $n - 1$  players. In simple term comparisons, this means that we need a minimum of  $n - 1$  games.

There are two types of Tournament Trees:

1. **Winner tree:** A Winner Tree can be defined as the complete Binary Tree, where every node characterizes the smaller or greater of its two children. As in the case of the Binary Tree, it starts with the root, so root becomes the representation of the smaller or greater node. The Tournament Tree winner can be found by looking at the smallest or greatest  $n$  key in all the sequences. The time complexity of this tree is  $O(\log n)$ .
2. **Loser tree:** A Loser Tree can be defined as the complete Binary Tree with  $n$  external nodes and  $n - 1$  internal nodes. The loser is stored in the internal nodes of the tree. The root node at  $[0]$  is the winner of the tournament. The corresponding node is the loser node.

**Uses of tournament tree:** The following are the uses of this type of tree:

1. Finding the smallest and largest element in the array.
2. Sorting.
3. Merging M-way. (M-way merge is like merging M sorted arrays to get single sorted array).



What are the benefits of using a Tournament Tree?

### 17.3.2.3 Graph Structure

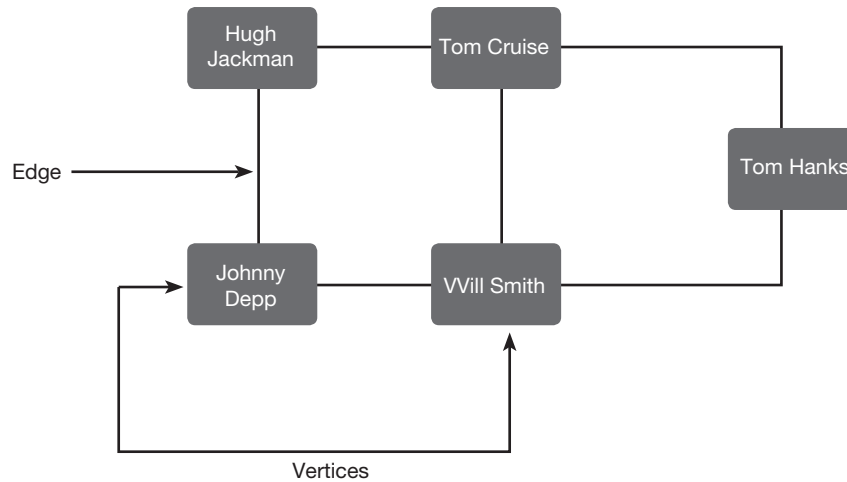
Graph data structure stores connected data where nodes are connected to each other in a network fashion. Think of a social media platform. You are connected to your friend, your friend connects back to you, there is someone connected to your friend, there may be someone whom you also know connected to your friend, and so on and so forth. This is how graph is setup. A

graph contains vertices and edges. The vertex represents an entity like people and the edge is the relationship between those entities.



Can a graph be useful to store a dictionary values?

Figure 17.8 shows a graph, edges, and vertices.

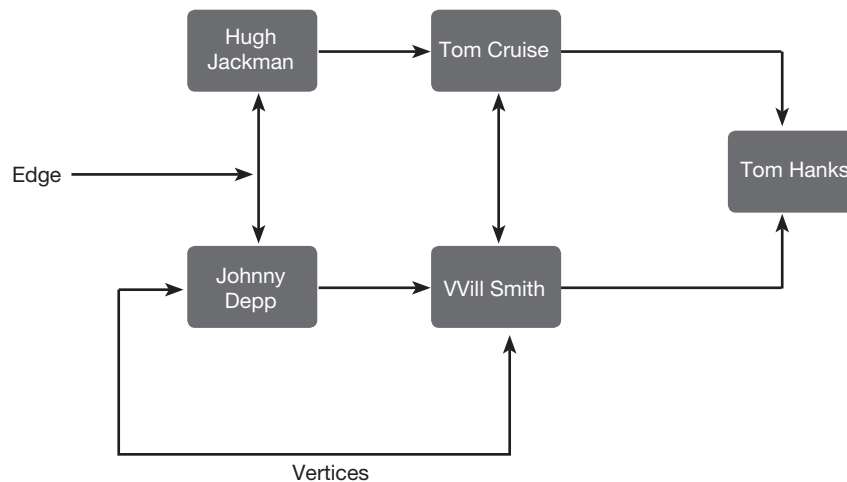


**Figure 17.8** Example of a graph.

In the above example, you can see that the nodes are named as Johnny Depp, Will Smith, Hugh Jackman, Tom Cruise, and Tom Hanks. These are entities and the connection between these nodes are edges.

There are a few variations to the graph data structure of Figure 17.8 . These are discussed in the following subsections.

### 17.3.2.3.1 Directed Graph



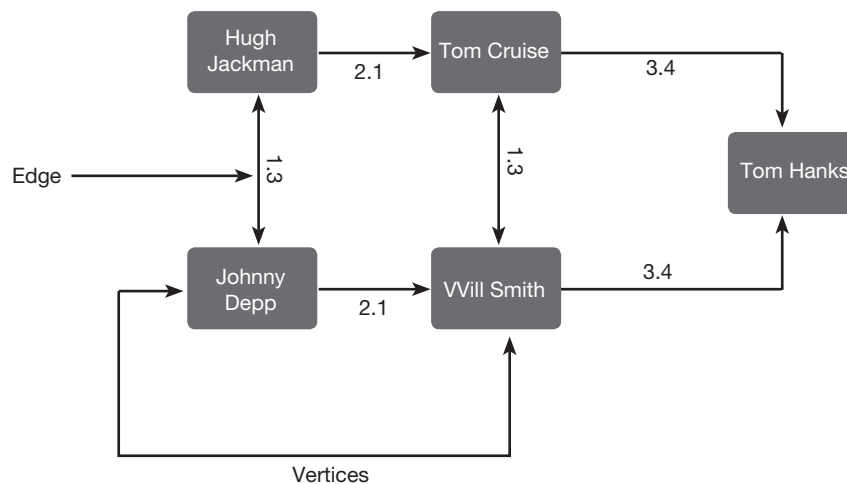
**Figure 17.9** Example of directed graph.

In the above example shown in Figure 17.8, we do not see any direction. That is, the nodes are just connected and do not carry any specific direction. If we add direction to the edges, then the graph becomes a directed graph. It means that the edges

point the relationship direction between nodes. In the social media example, this could be used to show who sends a friendship request. These edges can carry bidirectional relationship as well as shown in 17.9. Bidirectional relationship exists between two nodes which are connected both ways. In other words, the connection between node A and node B is taking place from node A to node B and vice versa.

### 17.3.2.3.2 Weighted Graph

We have now seen a graph with edges and one with direction-based edges. If these edges carry relative weight, they become a weighted graph as shown in Figure 17.10. In the social media example, each connection carries direction, like the connection between you and your friend A. If we add the number of years you know your friend to the edge, then that becomes the weight of the relationship.



**Figure 17.10** Example of weighted graph.

### 17.3.2.4 Graph Representations

A graph has two commonly used representations – adjacency matrix and adjacency list. It also has other representations that are less commonly used – incidence matrix and incidence list.

#### 17.3.2.4.1 Adjacency Matrix

An adjacency matrix is a square matrix. In simple terms, as shown in Figure 17.11, an adjacency matrix shows if the elements in the graph are next to each other. The number of vertices in the graph defines the dimensions of the graph. This matrix has simple representation, which contains values of 0 or 1. A value of 1 indicates that there is an adjacency between row and column. And a value of 0 indicates that there is no adjacency between row and column. Let us take our example and see how it can be represented as a graph.



Can an adjacency matrix be useful to define edge weight?

This is the easiest representation to implement and follow. The complexity of edge removal is  $O(1)$  time and the edge find between two vertices can also be done in  $O(1)$  time. However, the adjacency matrix takes a larger space of  $O(V^2)$  time (here,  $V$  is vertex set) even though it may contain lesser number of edges. The edge addition takes  $O(V^2)$  time.

	Hugh Jackman	Tom Cruise	Tom Hanks	Johnny Depp	Will Smith
Hugh Jackman	0	1	0	1	0
Tom Cruise	1	0	1	0	0
Tom Hanks	0	1	0	0	1
Johnny Depp	1	0	0	0	1
Will Smith	0	1	1	1	0

**Figure 17.11** Example of adjacency matrix.

#### 17.3.2.4.2 Adjacency List

As shown in Figure 17.12, an adjacency list is simply an array of lists. The number of vertices in the graph determines the size of the array.  $\text{Array}[i]$  represents the  $i$ th vertex's adjacent vertices. This list can also represent a weighted graph. The weights of edges in the weighted graphs are represented as lists of pairs. The adjacency list representation takes less space  $O(|V|+|E|)$ , where  $V$  is vertex set and  $E$  is edges. In a worst-case scenario, it may take  $O(V^2)$  space, even though there may be  $C(V, 2)$  number of edges. Adding a vertex is relatively easier than in an adjacency matrix. However, the edge find process between two nodes may be expensive, such as  $O(V)$  times.



**Figure 17.12** Example of adjacency list.

### 17.3.2.4.3 Incidence Matrix

Incidence matrix is useful to show the relationship between objects of two classes. This matrix is built in a way that it contains one row for each element of first class and one column for each element of second class. If first class row  $x$  and second class row  $y$  are related, it contains 1 in row  $x$  and column  $y$  which is called incident in this context and it contains 0 if they are not related.

Incidence matrices are frequently used in graph structure, such as undirected and directed graphs.

1. **Undirected graph:** Let us take the example shown in Figure 17.13 of an undirected graph,  $X$ . This graph contains vertices and edges; say,  $n$  number of vertices and  $m$  number of edges. So, this matrix  $Y$  of graph  $X$  is of  $n \times m$  matrix. According to the incidence matrix case, this matrix will have  $Y_{i,j} = 1$  if the vertex  $v_i$  and edge  $e_j$  are related also called incident and 0 if they are not related.

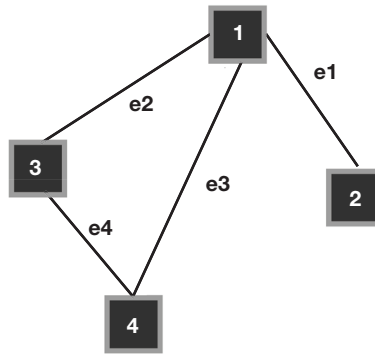


Figure 17.13 Undirected graph.

Table 17.3 shows the matrix based on the undirected graph shown in Figure 17.13.

Table 17.3 Incidence matrix for undirected graph

Sl. No.	e1	e2	e3	e4
1	1	1	1	0
2	1	0	0	0
3	0	1	0	1
4	0	0	1	1

2. **Directed graph:** As the name suggests, directed graph is one which contains vertices connected by edges. These edges carry a direction with them. The incidence matrix of a directed graph,  $D$ , is shown in Figure 17.14. It contains  $n$  number of vertices and  $m$  number of edges of matrix  $M$ . So, matrix  $M$  is an  $n \times m$  matrix, as shown in Table 17.4. This matrix content is set based on the directed relationship. For example, if edge  $e_j$  leaves vertex  $v_i$  then position  $M_{i,j} = -1$ , if it enters vertex  $v_i$  then 1 and 0 if it does not enter.

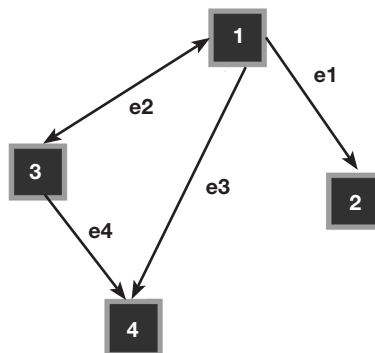


Figure 17.14 Directed graph.

Table 17.4 shows the matrix based on the directed graph shown in Figure 17.14.

**Table 17.4** Incidence matrix for directed graph

Sl. No.	e1	e2	e3	e4
1	-1	1	-1	0
2	1	0	0	0
3	0	1	0	-1
4	0	0	1	1

## Summary

Data structure is an important concept of programming. In any programming language, you will encounter at least one data structure that you will need to complete your program. There are various types of data structures. Some are very common, while others are rarely used but very useful in solving complex problems.

In this chapter, we have learned the following:

1. What is a data structure?
2. How to use data structure in programs.
3. Linear and non-linear data structures.
4. Benefits of using Binary Tree.
5. Difference between tree and graph.
6. Directed and weighted graph.
7. Adjacency matrix and adjacency list.

In Chapter 18, we will learn about lambdas and functional programming. We will spend time understanding lambdas and explore the ways to use it in a program. We will explore the use of functional programming.

## Multiple-Choice Questions

1. Which of the following has elements arranged to each other?
  - (a) Linear Data Structure
  - (b) Non-Linear Data Structure
  - (c) Both (a) and (b)
  - (d) None of the above
2. Which one of the following data types is utilized only for positive values in data structure?
  - (a) Arrays
  - (b) Unsigned
  - (c) Signed
  - (d) Boolean
3. What is the minimum number of nodes that a binary tree can possess?
  - (a) Two
  - (b) One
  - (c) Zero
  - (d) None
4. You can access an array by referring to the indexed element within the array.
  - (a) True
  - (b) False
5. Which of the following trees has each internal node containing an operator and each leaf node containing an operand?
  - (a) Winner Tree
  - (b) Loser Tree
  - (c) Expression Tree
  - (d) Binary Search Tree

## Review Questions

1. What is data structure?
2. How do we use data structure?
3. What is linear data structure?
4. What is non-linear data structure?
5. What is the difference between linear and non-linear data structure?

6. Give at least two scenarios on when to use linear and when to use non-linear data structure.
7. Define tree data structure.
8. What are the frequently used subsets of tree data structure?
9. What is the search complexity of Binary Search Tree?
10. What is Expression Tree?
11. What is graph structure?
12. Give one real-life example of graph structure in use.
13. What is directed graph?
14. What is weighted graph?
15. How do you create adjacency matrix?
16. How do you create adjacency list?
17. Which graph representation is faster in finding the edge between two nodes?
18. What is the complexity in removing edge in adjacency matrix?

## Exercises

---

1. Plot your Facebook account data and figure out your personal graph like your friends, their friends, common friends, etc. Design a graph and add weight to it.
2. Create an adjacency matrix and adjacency list for the graph in Question 1.
3. Create a chart to show the differences, advantages, and disadvantages between linear and non-linear data structure.

## Project Idea

---

Design your own social media platform. Create pages to add users and let them search the platform to look for connections. Write a detailed plan on how to store and retrieve data. Figure out the best possible data structure for

this problem and design one. Collect as much as data and run various algorithms on it to learn how you can extend functionalities for your social network.

## Recommended Readings

---

1. Allen B. Downey. 2017. *Think Data Structures: Algorithms and Information Retrieval in Java*. O'Reilly Media: Massachusetts
2. James Cutajar. 2018. *Beginning Java Data Structures and Algorithms: Sharpen your problem solving skills by learning core computer science concepts in a pain-free manner*. Packt: Birmingham
3. Mr Kotiyana. 2018. *Introduction to Data Structures and Algorithms in Java*. [Independently Published]
4. Suman Saha and Shailendra Shukla. 2019. *Advanced Data Structures: Theory and Applications*. Chapman and Hall/CRC: London